Ing. Inversa De troyanos I

Bueno vengo a compartir con ustedes lo que en mi parecer es una grandiosa clase de "ing. inversa" en los Troyanos de mi amigo Bloodday (Un gran saludo si ves esto, haber cuando te pasas por "ndtctbls" XD)

---

como el titulo del post lo dice, esta será la primera clase de la serie "ing. Inversa de troyanos" en la que aprenderemos a ver que hace exactamente X troyano, como modificar ciertas cosas a nuestro favor, y si es posible, agregar nuevas funciones, pero esto lo decidirá el tiempo.

En nuestras primeras clases... empezaremos con algunas definiciones, veremos lo que hacen las instrucciones de ensamblador, y algunas APIs de Windows, les diría que se leyeran el "Windows API for programmers" pero hay algunas APIs que no esta ahí (por ejemplo busquen DebugActiveProcessStop y no les aparecerá) así que para las APIs que no sepamos que hacen, usaremos el Puto google (con esto también me incluyo por que hay muchas APIs que no se que hacen...)

Empecemos con las definiciones...

Registros: los registros, son algo así como "variables" que usa el procesador, y hay varios de ellos, y cada uno tiene una tarea especifica... nosotros nos concentraremos en los que nos muestra el ollydbg, que son EAX, EBX, ECX, EDX, EBP, ESP, ESI, EDI y EIP. Pero que carajos significa eso?

Pues bien, la “E” significa EXTENDIDO, pero por que extendido? Pues sencillo, antes se trabajaba con 16 bits y los registros eran AX, BX, CX, DX, BP, SP, SI, DI e IP, y para poder trabajar con 32 bits se “inventaron” estos registros extendidos.

Bueno, para dar una explicación sencilla diremos que los registros que terminan en “X” (EAX, EBX, ECX y EDX) son registros de uso general, pero además de ello también tienen una tarea específica que cumplir. Los registros que terminan en “P” (EIP, EBP y ESP) son registros que llevan un Puntero (Pointer en ingles) es por eso que terminan en “P” y Finalmente los dos registros que terminan en “I” (ESI y EDI) son registros de Índice.

Como ya hemos dicho, cada registro tiene una función específica, y la segunda letra de su nombre nos indica la función asignada a cada registro, de esta manera la “A” de EAX, quiere decir que es el registro Acumulador, por ser aquí donde se guarden los resultados de ciertas operaciones (div y mul por ejemplo) y donde la mayoría de las APIs retornen algún valor.

La “B” de EBX, y EBP nos indica que son registros de Base, en el caso de EBX además de su uso como registro general, suele utilizarse para direccionar el acceso a datos situados en la memoria. Y EBP es utilizado para direccionar el acceso a datos situados dentro del espacio ocupado por la pila, aunque también puede ser de uso general.

La “C” de ECX, nos indica que es el registro Contador (Counter en ingles) y es así, por que es el registro utilizado para hacer bucles, además de ser un registro de uso general.

La “D” de EDX nos indica que es un registro de Datos, y es así por ser utilizado en instrucciones de entrada y salida, pero además de esto, también es utilizado en conjunto con EAX para formar números de mas de 32 bits en operaciones como multiplicar (mul) o dividir (div)

La “S” de ESP viene dada por que como ya se dijo lleva un puntero, y este es el de la Pila (Stack en ingles) es decir que apunta hacia el primer valor de ésta.

La “I” de EIP significa Instrucción, es decir que este registro nos dice el Puntero o la dirección de la siguiente instrucción a ejecutar.

Y por ultimo los registros de Índice ESI y EDI, pues estos registros son utilizados por ciertas operaciones como origen (ESI) y destino (EDI) por ejemplo para copiar un bloque de bytes.

API: Las APIs (Application Programming Interface) son rutinas "prefabricadas" destinadas a facilitar la labor del Programador y al mismo tiempo a reducir la extensión de los programas, porque no hace falta incluir estas Rutinas en el ejecutable, ya que están incluidas en librerías de Windows.

Flags: son Bits que se ponen a uno o cero según la instrucción que ejecutemos, y hay instrucciones que hacen una cosa u otra según el estado de alguno de ellos (saltos condicionales por ejemplo)

Instrucciones del lenguaje ASM (sacadas de “ASM por Caos Reptante” por que me parece que mejor explicadas no podrían estar)

Vamos a ver la mayoría, pero no todas, he dejado de lado las que me han parecido menos importantes.

And El resultado es 1 si los dos operandos son 1, y 0 en cualquier otro caso.
1 and 1 = 1
1 and 0 = 0
0 and 1 = 0
0 and 0 = 0
Ejemplo: 1011 and 0110 = 0010

Or El resultado es 1 si uno o los dos operandos es 1, y 0 en cualquier otro caso.
1 or 1 = 1
1 or 0 = 1
0 or 1 = 1
0 or 0 = 0
Ejemplo: 1011 or 0110 = 1111

Xor El resultado es 1 si uno y sólo uno de los dos operandos es 1, y 0 en cualquier otro caso

1 xor 1 = 0
1 xor 0 = 1
0 xor 1 = 1
0 xor 0 = 0
Ejemplo: 1011 xor 0110 = 1101

Not Simplemente invierte el valor del único operando de esta función
Not 1 = 0
Not 0 = 1
Ejemplo: not 0110 = 1001

Estos ejemplos los hemos visto con números binarios. Como estas operaciones se hacen bit a bit, si quisiéramos hacer una operación entre números en formato hexadecimal, que es lo más corriente, deberíamos pasarlos antes a binario. O bien utilizar la calculadora de Windows en formato científico.

Nop (No Operation)

Esta instrucción es de gran utilidad: no hace nada. Su verdadera utilidad reside en rellenar los huecos provocados por la eliminación de instrucciones o su substitución por instrucciones de menor longitud. Su nombre da origen al verbo "nopear" que tan a menudo utilizan los chicos malos ;-) Hay corrientes filosóficas dentro del cracking que sostienen que poner nops es una cochinada.

-y aquí tengo que hacer una pausa para reseñar que la instrucción **NOP** NO EXISTE pues si vemos su opcode (bytes de la instrucción) es 90, y este opcode pertenece a la instrucción **XCHG EAX,EAX** pero como es obvio que esto no modifica nada, los debuggers y desensambladotes usan este “convencionalismo”

Push (Push Word or Doubleword Onto the Stack)

Esta instrucción resta del registro ESP, la longitud de su único operando que puede ser de tipo Word o doubleword (4 u 8 bytes), y a continuación lo coloca en la pila. (en pocas palabras pone lo que le indiques de primero en la pila)

Pop (Pop a Value from the Stack)

Es la inversa de la anterior, es decir que incrementa el registro ESP y retira el valor disponible de la pila y lo coloca donde indica el operando.

Pushad (Push All General-Purpose Registers)

Estas instrucciones guardan el contenido de los registros en la pila en un orden determinado. Así pues, pushad equivale a: push EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Popad (Pop All General-Purpose Registers)

Estas instrucciones efectúan la función inversa de las anteriores, es decir, restituyen a los registros los valores recuperados de la pila en el orden inverso al que se guardaron.

Así popad equivale a: pop EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.

Pushfd (Push EFLAGS Register onto the Stack)
Popfd (Pop Stack into EFLAGS Register)

Estas parejas de instrucciones colocan y retiran de la pila el registro de flags.

mov (Move)

Esta instrucción tiene dos operandos. Copia el contenido del operando de origen (representado en segundo lugar) en el de destino (en primer lugar), y según el tipo de estos operandos adopta formatos distintos. He aquí unos ejemplos:
Código:
```
6689C8 mov ax, cx
8BC3 mov eax, ebx
8B5BDC mov ebx, dword ptr [ebx-24]
893438 mov dword ptr [eax+edi], esi
```
movsx (Move with Sign-Extension)

Copia el contenido del segundo operando, que puede ser un registro o una posición de memoria, en el primero (de doble longitud que el segundo), rellenándose los bits sobrantes por la izquierda con el valor del bit más significativo del segundo operando. Aquí tenemos un par de ejemplos:
Código:
```
33C0 xor eax, eax
BB78563412 mov ebx, 12345678
0FBFC3 movsx eax, bx EAX=00005678
33C0 xor eax, eax
BBCDAB3412 mov ebx, 1234ABCD
0FBFC3 movsx eax, bx EAX=FFFFABCD
```
Vemos como en el primer ejemplo los espacios vacíos se rellenan con ceros y en el segundo con unos.

movzx (Move with Zero-Extend)

Igual a movsx, pero en este caso, los espacios sobrantes se rellenan siempre con ceros. Veamos como el segundo ejemplo de la instrucción anterior da un resultado distinto:
Código:
```
33C0 xor eax, eax
BBCDAB3412 mov ebx, 1234ABCD
0FB7C3 movzx eax, bx EAX=0000ABCD
```
lea (Load Effective Address)

Similar a la instrucción mov, pero el primer operando es un registro de uso general y el segundo una dirección de memoria. Esta instrucción es útil sobre todo cuando esta dirección de memoria responde a un cálculo previo.
Código:
```
8D4638 lea eax, dword ptr [esi+38]
```
xchg (Exchange Register/Memory with Register)

Esta instrucción intercambia los contenidos de los dos operandos.
Código:
```
87CA xchg edx, ecx
```

```
870538305100 xchg dword ptr [00513038], eax
8710 xchg dword ptr [eax], edx
```
En el primer ejemplo, el contenido del registro ECX, se copia en el registro EDX, y el contenido anterior de EDX, se copia en ECX. Se obtendría el mismo resultado con:
Código:
```
51 push ecx
52 push edx
59 pop ecx
5A pop edx
```
La instrucción pop ecx toma el valor que hay en la pila y lo coloca en el registro ECX, pero como podemos ver por la instrucción anterior push edx, este valor, procedía del registro EDX. Luego se coloca en EDX el valor procedente de ECX.

bswap (Byte Swap)

Esta instrucción es para empleo exclusivo con registros de 32 bits como único parámetro. Intercambia los bits 0 a 7 con los bits 24 a 31, y los bits 16 a 23 con los bit 8 a 15.
Código:
```
B8CD34AB12 mov eax, 12AB34CD EAX=12AB34CD
0FC8 bswap eax EAX=CD34AB12
```
Inc (Increment by 1) / dec (Decrement by 1)

Estas dos instrucciones incrementan y decrementan respectivamente el valor indicado en su único operando.
Código:
```
FF0524345100 Inc dword ptr [00513424]
FF0D24345100 dec dword ptr [00513424]
40 inc eax
4B dec ebx
```
En los dos primeros ejemplos, se incrementaría y decrementaría el valor contenido en los cuatro bytes situados a partir de la dirección 513424.

add (Add)

Esta instrucción suma los contenidos de sus dos operandos y coloca el resultado en el operando representado en primer lugar.
Código:
```
02C1 add al, cl AL + CL -> AL
01C2 add edx, eax EDX + EAX -> EDX
8345E408 add dword ptr [ebp-1C], 0000008 dword ptr [EBP-1C]+ 8 > [EBP-
1C]
```
En el tercer ejemplo, como el resultado se coloca siempre en el primer operando, no sería aceptada por el compilador una instrucción como add 00000008, dword ptr [ebp-1C]

adc (Add with Carry)

Esta instrucción es similar a la anterior, con la diferencia de que se suma también el valor del flag de acarreo. Se utiliza para sumar valores mayores de 32 bits. Supongamos que queremos sumar al contenido de los registros EDX:EAX (EDX=00000021h y EAX=87AE43F5), un valor de más de 32 bits (3ED671A23). Veamos como se hace:
Código:

```
Add eax, ED671A23 EAX=75155E18 (87AE43F5+ED671A23) CF=1
adc edx, 0000003 EDX=25h (21h+3+1)
```
sub (Subtract)

Esta instrucción resta el contenido del segundo operando del primero, colocando el resultado en el primer operando.
Código:
```
83EA16 sub edx, 00000016 EDX - 16 -> EDX
29C8 sub eax, ecx EAX - ECX -> EAX
2B2B sub ebp, dword ptr [ebx] EBP - dword ptr [EBX] -> EBP
```
sbb (Integer Subtraction with Borrow)

Esta instrucción es una resta en la que se tiene en cuenta el valor del flag de acarreo. Supongamos que del contenido de los registros EDX:EAX después del ejecutado el ejemplo de la instrucción adc (EDX=00000025h y EAX=75155E18), queremos restar el valor 3ED671A23. El resultado es el valor que tenían inicialmente los dos registros en el ejemplo citado:
Código:
```
Sub eax, ED671A23 EAX=87AE43F5 (75155E18-ED671A23) CF=1
sbb edx, 0000003 EDX=21h (25h-3-1)
```
mul (Unsigned Multiply) / **imul** (Signed Multiply)

Estas dos instrucciones se utilizan para multiplicar dos valores. La diferencia más importante entre las dos, es que en la primera no se tiene en cuenta el signo de los factores, mientras que en la segunda sí. Como veremos en algunos ejemplos, esta diferencia se refleja en los valores de los flags.

En la instrucción mul, hay un solo operando. Si es un valor de tamaño byte, se multiplica este valor por el contenido de AL y el resultado se guarda en EAX, si el valor es de tamaño word (2 bytes), se multiplica por AX, y el resultado se guarda en EAX y finalmente, si el valor es de tamaño dword (4bytes), se multiplica por EAX y el resultado se guarda en EDX:EAX. O sea, que el espacio destinado al resultado siempre es de tamaño doble al de los operandos.
Código:
```
F7E1 mul ecx EAX*ECX -> EDX:EAX
F72424 mul dword ptr [esp] EAX*[ESP] -> EDX:EAX
```
En la instrucción imul, hay también una mayor variedad en el origen de sus factores. Además de la utilización de los registros EAX y EDX, así como de sus subdivisiones, pueden especificarse otros orígenes y destinos de datos y puede haber hasta tres operandos. El primero, es el lugar donde se va a guardar el resultado, que debe ser siempre un registro, el segundo y el tercero son los dos valores a multiplicar. En estos ejemplos vemos como estas instrucciones con dos o tres operandos, tienen el mismo espacio para el resultado que para cada uno de los factores:
Código:
```
F7EB imul ebx EAX x EBX -> EDX:EAX
696E74020080FF imul ebp, dword ptr [esi+74], FF800002 [ESI+74] x
FF800002 -> EBP
0FAF55E8 imul edx, dword ptr [ebp-18] EDX x [EBP-18] -> EDX
```
Veamos finalmente la diferencia entre la multiplicación con signo y sin él:
Código:
```
66B8FFFF mov ax, FFFF AX=FFFF
66BBFFFF mov bx, FFFF
66F7E3 mul bx AX=0001 OF=1 CF=1
```

```
66B8FFFF mov ax, FFFF
66F7EB imul bx AX=0001 OF=0 CF=0
```
En la primera operación, como se consideran los números sin signo, se ha multiplicado 65535d por 65535d y el resultado ha sido 1. Debido a este resultado "anómalo", se han activado los flags OF y CF. En cambio, en la segunda operación, en la que se toma en cuenta el signo, se ha multiplicado -1 por -1 y el resultado ha sido 1. En este caso no se ha activado ningún flag. Otro ejemplo:
Código:
```
B8FFFFFF7F mov eax, 7FFFFFFF
BB02000000 mov ebx, 00000002
F7E3 mul ebx EAX=FFFFFFFE OF=0 CF=0
B8FFFFFF7F mov eax, 7FFFFFFF
F7EB imul ebx EAX=FFFFFFFE OF=1 CF=1
```
Esta vez, en el primer caso, el resultado ha sido correcto, porque 2147483647d multiplicado por dos ha dado 4294967294d, por tanto, no se ha activado ningún flag. Pero en el segundo caso, teniendo en cuenta el signo, hemos multiplicado 2147483647d por dos y nos ha dado como resultado -2. Ahora si se han activado los flags.

div (Unsigned Divide) / **idiv** (Signed Divide)

El caso de la división es muy parecido al de la multiplicación. Hay dos instrucciones: div para números en los que no se considere el signo e idiv para números que se consideren con signo. El dividendo está formado por una pareja de registros y el divisor es el único operando. He aquí varios ejemplos de una y otra instrucción:
Código:
```
66F7F3 div bx DX:AX : BX -> AX resto -> DX
F7F3 div ebx EDX:EAX : EBX -> EAX resto -> EDX
F77308 div dword ptr [ebx+08] EDX:EAX : [EBX+8] -> EAX resto -> EDX
F7F9 idiv ecx EDX:EAX : ECX -> EAX resto -> EDX
```
Ahora, como hemos hecho con la multiplicación, vamos a ver el diferente resultado que se obtiene empleando una u otra instrucción:
Código:
```
33D2 xor edx, edx
66B80100 mov ax, 0001
66BBFFFF mov bx, FFFF
66F7F3 div bx AX=0000 DX=0001
33D2 xor edx, edx
66B80100 mov ax, 0001
66F7FB idiv bx AX=FFFF DX=0000
```
En el primer caso, al no considerar el signo de los números, se ha dividido 1 por 65535, que ha dado un cociente de 0 y un resto de 1. En el segundo caso, se ha dividido -1 por 1, lo que ha dado un cociente de -1 y un resto de 0. No ha habido overflow ni acarreo en ninguno de los dos casos.

xadd (Exchange and Add)

Intercambia los valores de los dos operandos y los suma, colocando el resultado en el primer operando. El primer operando puede ser un registro o una posición de memoria, pero el segundo sólo puede ser un registro.
Código:
```
C703CDAB3412 mov dword ptr [ebx], 1234ABCD
```
En la dirección indicada por EBX tendremos el valor CD AB 34 12. Vemos el valor que hemos puesto en la memoria invertido, porque el paso del valor de un registro a la

memoria y viceversa se hace empezando por el último byte y terminando por el primero.
Código:
```
B8CD34AB12 mov eax, 12AB34CD
B934120000 mov ecx, 00001234
0FC1C8 xadd eax, ecx
```
EAX contiene el valor 12AB4701 (12AB34CD+1234) y ECX el valor 12AB34CD.
Código:
```
B934120000 mov ecx, 00001234
0FC10B xadd dword ptr [ebx], ecx
```
La dirección indicada por EBX contiene el valor 01 BE 34 12 (1234ABCD+1234) y el registro ECX el valor 1234ABCD.

neg (Two's Complement Negation)

Esta instrucción tiene la finalidad de cambiar de signo el número representado por su único operando, mediante una operación de complemento a dos.
Código:
```
B81C325100 mov eax, 0051321C
F7D8 neg eax EAX=FFAECDE4

neg 0000 0000 0101 0001 0011 0010 0001 1100 = 0051321C
       1111 1111 1010 1110 1100 1101 1110 0011
  + 1
       1111 1111 1010 1110 1100 1101 1110 0100 = FFAECDE4
```
CMP (Compare Two Operands)

La comparación entre dos valores es en realidad una resta entre ambos. Según cual sea el resultado, podemos saber si los valores son iguales y en caso contrario, cual de ellos es el mayor. Así, se podría utilizar la instrucción sub ecx, ebx para comparar el resultado de estos dos registros, sin embargo el hacerlo así tiene el problema de que el resultado de la resta se colocaría en el registro ECX, cuyo valor anterior desaparecería. Para evitar este problema el programador dispone de la instrucción cmp.

Esta instrucción resta el segundo operando del primero. El resultado no se guarda en ningún sitio, pero según cual sea este resultado, pueden modificarse los valores de los flags CF, OF, SF, ZF, AF y PF. Es en base al estado de estos flags, que se efectúa o no el salto condicional que suele acompañar a esta instrucción. Veremos esta instrucción en los ejemplos de saltos condicionales.

Bueno, eso es todo por esta clase, si tienen preguntas, háganlas por acá o por el Chat, para la próxima veremos algunas APIs, y nos introduciremos en el Olly para aprender a utilizarlo y ver las posibilidades que nos da.

Para los que quieran leer el documento “ASM por Caos Reptante” se los dejo adjunto.

ASM por CaoS ReptantE.rar

PD: disculpen lo largo, y faltaron los saltos, pero será para la próxima clase